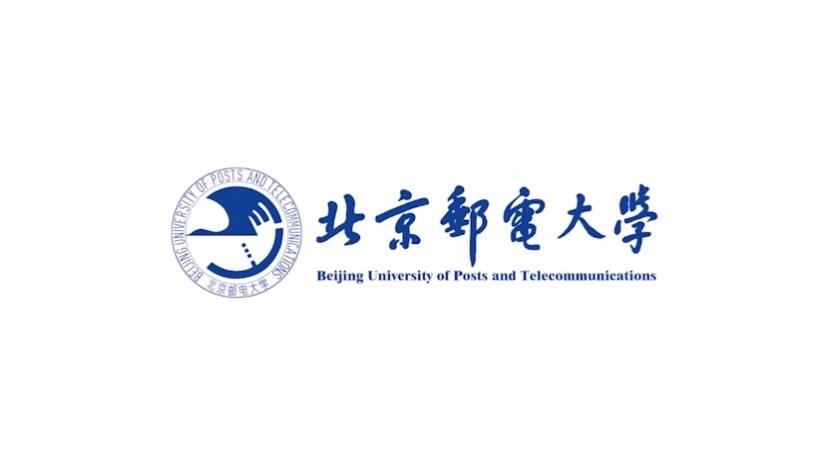
**算法设计与分析实验报告**



实验题目： 利用FFT算法改进大整数乘法的算法效率

姓名： 陈俊卉

学号： 2020212256

日期： 2022.10.17

# 一、实验环境

(列出实验的操作环境，如操作系统，编程语言版本等，更多信息可视各自实际情况填写)

1. 操作系统：windows 10
2. 编程语言：c++
3. 编程工具：vscode及其组件

# 二、实验内容

具体要求请参照实验指导书中各实验的“实验内容及要求”部分。

(示例：1.描述你对实验算法的设计思路;2.给出算法关键部分的说明以及代码实现截图；3.对测试数据、测试程序(没有要求则非必须)进行说明，如测试覆盖程度，最好最坏平均三种情况等等，并给出测试结果截图等信息)

### 1.算法的设计与实现

##### **(1) 分治法实现大整数乘法**

#include<iostream>  
#include<cmath>  
using namespace std;  
​  
int sign(long long int num){  
    if (num > 0){return 1;}  
    else {return -1;}  
}  
​  
​  
long long int integersMultiplication(long long int x, long long int y, long long int len\_x, long long int len\_y){  
    if (x == 0 || y == 0){  
        return 0;  
   }  
​  
    else if (len\_x == 1 || len\_y == 1){  
        return x \* y;  
   }  
    else{  
        // 推导：  
        // x = a \* 10^(len\_x / 2) + b       y = c \* 10^(len\_y / 2) + d  
        // xy = ac \* 10^(len\_x / 2 + len\_y / 2) + (ad \* 10^(len\_x / 2) + bc \* 10^(len\_y / 2)) + bd  
        // F1 = ac F2 = bd F3 = (a \* 10^(len\_x / 2) - b) \* (d - c \* 10^(len\_y / 2))  
        // 可化简为  
        // xy = F1 \* 10^(len\_x / 2 + len\_y / 2) + (F3 + F1 \* 10^(len\_x / 2 + len\_y / 2) + F2) +F2  
​  
        int len\_b = len\_x / 2;  
        int len\_a = len\_x - len\_b;  
        int len\_d = len\_y / 2;  
        int len\_c = len\_y -len\_d;  
​  
        long long int a = (long long int)(x / pow(10, len\_b));  
        // 带符号，不用再多考虑符号问题  
        long long int b = (long long int)(x % (long long int)pow(10, len\_b));  
        long long int c = (long long int)(y / pow(10, len\_d));  
        // 带符号，不用再多考虑符号问题  
        long long int d = (long long int)(y % (long long int)pow(10, len\_d));  
        long long int F1 = integersMultiplication(a, c, len\_a, len\_c);  
        long long int F2 = integersMultiplication(b, d, len\_b, len\_d);  
        long long int F3 = integersMultiplication((long long int)(a \* pow(10, len\_b) - b), (long long int)(d - c \* pow(10, len\_d)), len\_b, len\_d);  
      
        return (long long int)(F1 \* pow(10, len\_b + len\_d) + (F3 + F1 \* pow(10, len\_b + len\_d) + F2) + F2);  
      
   }  
​  
​  
​  
}  
​  
int main(){  
    // 输入数据  
    long long int x = 98765;  
    long long int y = -6800;  
    cout << integersMultiplication(x, y, 5, 4);  
}

###### **② 原理：**

这是经过改进的分治法。未经过改进的分治法与竖式法都是O(n^2).**上述代码为了满足任意情况（任意长度的两个数据相乘），采用了与PPT不一样的改善方法**，具体如下所示：

对于大整数x、y，令

故：

设：

则：

经过改进后，递归式为：

##### **(2) fft实现大整数乘法**

#include<iostream>  
#include<complex>  
using namespace std;  
​  
// 最大位数  
const int MAX = 1 << 20;  
// pi  
const double pi = acos(-1);  
​  
// 定义复数系数数组  
complex<double> a[MAX], b[MAX];  
// 定义a,b多项式长度，这里默认相同，为n（与分治法对齐）  
int n;  
int m;  
// 注意到fft与ifft区别仅有一个符号，所以写在一个函数内,使用i进行判别：i=1时为fft，i=-1时为ifft  
void fft\_or\_ifft(int len, complex<double> \*a, int i){  
    // 如果只有一项，则不需要再拆分，返回  
    if (len <= 1){return ;}  
    int mid = len >> 1;  
    // 定义奇数、偶数子数组  
    complex<double>\* A1 = new complex<double>[mid + 1];  
    complex<double>\* A2 = new complex<double>[mid + 1];  
    for (int i = 0; i <= len; i += 2){  
        A1[i >> 1] = a[i];  
        A2[i >> 1] = a[i + 1];  
   }  
    fft\_or\_ifft(mid, A1, i);  
    fft\_or\_ifft(mid, A2, i);  
    complex<double> w1(cos(pi / mid), i \* sin(pi / mid));  
    complex<double> w(1,0);  
    complex<double> x;  
    for (int i = 0; i < mid; i++){  
        x = w \* A2[i];  
        a[i] = A1[i] + x;  
        a[i + mid] = A1[i] - x;  
        w = w \* w1;  
   }  
}  
​  
int main(){  
    // 输入最高位  
    cin >> n;  
    cin >> m;  
    // 输入第一个式子  
    for (int i = 0; i <= n; i++){  
        double x;  
        cin >> x;  
        a[i].real(x);  
   }  
    // 输入第二个式子  
    for (int i = 0; i <= m; i++){  
        double x;  
        cin >> x;  
        b[i].real(x);  
   }  
​  
    // fft需要2的整数幂次; 默认两条式子长度相同  
    int len = 1 << int(fmax((int)ceil(log2(n + m)), 1));  
    fft\_or\_ifft(len, a, 1);  
    fft\_or\_ifft(len, b, 1);  
    for (int i = 0; i <= len; i++){  
        a[i] = a[i] \* b[i];  
   }  
    cout << endl;  
    fft\_or\_ifft(len, a, -1);  
​  
​  
    int\* res = new int[n + m + 1];  
    res[0] = 0;  
    // fft之后的结果复制到res  
    for (int i = 0; i <=n + m; i++){  
        cout << a[i].real() / len + 1e-6 <<" ";  
        res[i + 1] = a[i].real() / len + 1e-6;  
   }  
    cout << endl;  
​  
    // 处理  
    for (int i = n + m + 1; i > 0; i--){  
        if (res[i] >= 10){  
            int num = res[i] / 10;  
            // 防止最高位还有进位  
            res[i] = res[i] % 10;  
            res[i - 1] += num;  
       }  
   }  
​  
    cout << endl;  
    // for ( int i = 0; i <= n + m + 1;i++){  
    //     cout << res[i] << ' ';  
    // }  
​  
    for (int i = 0 ; i <= n + m + 1; i++){  
        if (i == 0 && res[i] == 0){continue;}  
        cout << res[i];  
   }  
}  
​

###### **② 原理：**

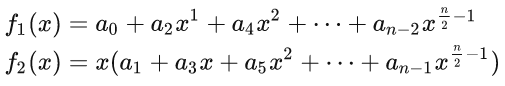
简而言之，就是先将多项式的系数表示经过fft变换为点值表示，两表达式的点值表示相乘之后再通过ifft的逆变换重新从点值表示转换为系数表示。实际上多项式相乘本身和大整数乘法的过程类似，只是最后的进位没有处理而已。在多项式变换的基础上，增加了进位处理就可以得到大整数相乘的fft。

时间复杂度的计算：

对于多项式

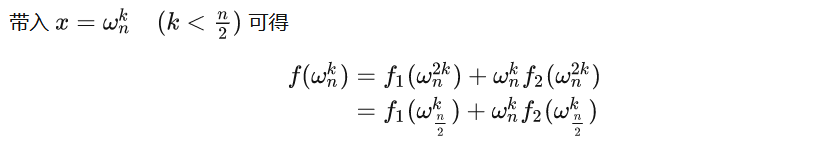


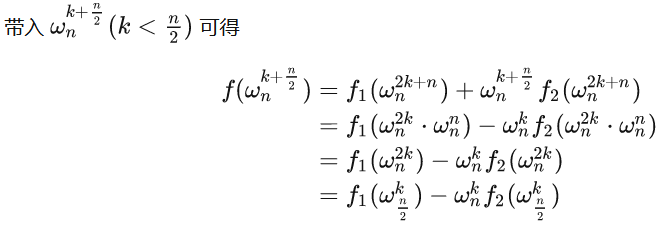
我们可以将其分为两个部分：



则有







则求解的时间为O(logn)，求解所有的时间为**O(nlogn)**.

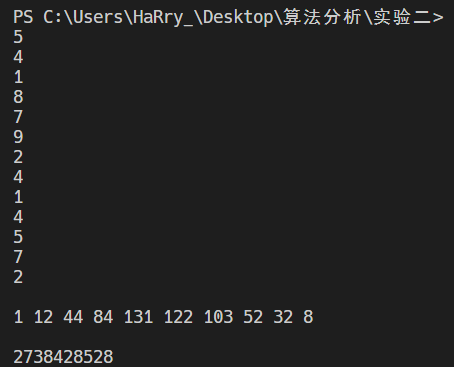
同样地，逆变换也是**O(nlogn).**

### 测试正确性

1. 分治法



1. FFT



输入输出解释：前两个数是两个数字的长度（5和4），第一个数是187924，第二个数是14572.输出的第一行是FFT多项式相乘的结果，第二行是乘法的结果。

### 最好/最坏情况

分治法的最好情况是两个数长度都为2的整数幂次。最坏情况是比2的整数幂次大1位。

FFT稳定，大概没有所谓的最好最坏情况。

# 三、出现问题及解决

(列出你在实验中遇到了哪些问题以及是如何解决的)

1. **分治法有大小限制（long long int）。**但碍于时间关系，没有将其改写成字符串输入输出的形式。
2. 分治法一开始的代码版本部分乘法答案不准。这是由于递归传参时的数字长度采用了除法造成的。一开始我写的是限定为2的幂次长度相等的输入，且长度参数只有一个。但这样在传参传n/2的时候，会导致长度不对应的问题。

# 四、总结

(对所实现算法的总结评价，如时间复杂度，空间复杂度，是否有能够进一步提升的空间，不同实现之间的比较，不同情况下的效率，通过实验对此算法的认识与理解等等)

本次实验，我熟悉了大整数乘法的分治法与FFT的原理推导与在大整数乘法的具体使用。虽然理解起来有些困难，但克服之后的成就感还是很满的。